

Approximate Two-Party Privacy-Preserving String Matching with Linear Complexity

Martin Beck
TU Dresden
Dresden, Germany
martin.beck1@tu-dresden.de

Florian Kerschbaum
TU Dresden
Dresden, Germany
florian.kerschbaum@tu-dresden.de

ABSTRACT

Consider two parties who want to compare their strings, e.g., genomes, but do not want to reveal them to each other. We present a system for privacy-preserving matching of strings, which differs from existing systems by providing a deterministic approximation instead of an exact distance. It is efficient (linear complexity), non-interactive and does not involve a third party which makes it particularly suitable for cloud computing. We extend our protocol, such that it mitigates iterated differential attacks proposed by Goodrich. Further an implementation of the system is evaluated and compared against current privacy-preserving string matching algorithms.

Categories and Subject Descriptors

K.4.4 [Computers and Society]: Electronic Commerce—*Security, Electronic Data Interchange*; K.4.1 [Computers and Society]: Public Policy Issues—*Privacy*

Keywords

approximate string matching, homomorphic encryption, bloom filter

1. INTRODUCTION

As technology for sequencing the human genome is developing at a fast pace and the number of sequenced genomes is rapidly growing, the need to process this highly personalized information in a privacy preserving way also increases. Several algorithms were presented which should protect the genomic information while it is being used across untrusted parties.

These protocols however are either interactive, match only exact strings or they require three parties to be involved. Our protocol is non-interactive, implements approximate string matching and does not require any third party. Our protocol is efficient and has linear complexity in computation and communication. It also withstands an iterated dif-

ferential attack proposed by Goodrich [1], that exploits the information gained by knowing the exact string distance (as proposed in other protocols).

The remainder of this paper is structured as follows. Section 2 gives an overview over basic concepts and related work. In section 3 the design will be presented and followed by section 4, which gives a security analysis of our system. Section 5 describes some implementation details and results in comparison to related systems. Section 6 concludes this work and points out further research directions.

2. RELATED WORK

Research into string matching algorithms is defined by a long list of proposed algorithms over many years and for many different problems. String matching itself is closely related to the distance between strings, which can be measured by a large variety of means, ranging from generic and simple solutions like the hamming distance [2] to more powerful algorithms like smith-waterman [3] solving local sequence alignment problems. A survey about current developments can be found in [4].

2.1 Approximate String Matching

As several tasks, for example checking whether a user profile is within a remote database, do not require the exact distance between two strings, data items or other entities, the notion of approximate matching was introduced to define levels of similarity, which in the most extreme way only output a single bit of information: if the input strings are similar or not. Due to these properties this class is called approximate string matching algorithms, which is not to be confused with the approximate string matching of [5], where the term “approximate” referred to the property of two strings of being close in distance.

2.2 Privacy-Preserving String Matching

Two of the applications for string comparison algorithms which are often used for motivation are calculating the distance of genome or protein sequences in bioinformatics and checking if a person is present in a remote database. As these topics by design deal with very personal information, which must not be given to third parties, the necessity to build privacy-preserving matching algorithms arose. As these were not sufficient to protect privacy due to information leakage given by the exact distance results, just obtained in a privacy preserving manner, combinations of the above mentioned approximation and the privacy-preserving computational steps were developed. A survey of recently published

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC’13 March 18–22, 2013, Coimbra, Portugal.

Copyright 2013 ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

algorithms together with benchmark results can be found in [6].

One of the more recent protocols introduced by Schnell et. al. [7] uses Bloom filters to represent strings and transforms the notion of distances between strings into distances between similar Bloom filters. We will also use Bloom filters as set representation for our genomic strings and build the matching protocol upon them. However, we use a two-party technique for comparing the Bloom filters and therefore do not need a trusted third party for comparing the strings. Furthermore, our protocol can be size-hiding, by choosing appropriate Bloom filter sizes, that are not proportional to the string length.

Alternatively, techniques from private set intersection (PSI) [8] could be used. However, revealing the content of the intersection is not appropriate for a privacy preserving protocol. Based on the security concerns protocols for private set intersection cardinality (PSI-CA) were developed [9]. This solution is still prone to the iterated differential attack described by Goodrich [1] as it leaks too much information.

Privacy-preserving protocols designed for approximate string comparisons can also be found in literature [10, 11], but rely on interactive techniques like oblivious transfers or secure computation. This excludes these protocols from off-line execution, e.g., in the cloud. Further [12] presents a more efficient solution, but which only matches exact strings. It could also be augmented using the techniques we present in this paper, like the usage of q -grams.

3. PROTOCOL DESIGN

To arrive at a privacy-preserving protocol, which is not vulnerable to the mentioned attack, we reduce the information gained by Alice down to a point where she only learns whether the set intersection cardinality is within an agreed range.

3.1 Bloom Filter Representation

A Bloom filter is a data structure fixed in size to which element representations can be added and on which member tests can be performed. Checking for an element is probabilistic due to the design of the filter.

Let b be an array of bits of length n and b_i the i th value within the array with $i \in [1, n]$. Further let $h_1() \dots h_k()$ be k hash functions, with uniformly distributed output in $[1, n]$. For initialization set $\forall i : b_i = 0$.

To add an element e to the filter, all k hash functions are evaluated and the results are taken as indexes for b to set these positions to one. Set $\forall j \in [1, k] : b_{h_j(e)} = 1$.

A member test is performed by also evaluating all k hash functions and checking the referenced positions in b . If at least one of the positions $b_{h_j(e)}$ is set to zero, the element has not been added to the Bloom filter before. If all bits are set to one, however, one cannot be sure if the exact element was inserted, or one or more different elements had these positions set to one.

Using these operations we are able to represent a set by adding all set elements to the filter. Depending on the filter parameters, the probability that a false-positive member test occurs, i.e. that we falsely identify an element as added to the filter before, is given by:

$$p = \left(1 - \left(1 - \frac{1}{n}\right)^{kl}\right)^k \quad (1)$$

Where $\left(1 - \frac{1}{n}\right)^{kl}$ is the probability that a single bit is still zero after l elements were added to the filter using k hash functions. To derive the required length of a Bloom filter given the false-positive rate and the number of elements to be inserted, the equation (1) can be transposed to:

$$l = \frac{-1}{(1 - p^{1/k})^{1/(k*n)} - 1} \quad (2)$$

3.2 String Matching Using Bloom Filters

A typical string comparison algorithm is the Levenshtein distance [13], which is often used as edit distance and describes the minimum number of insertions, deletions and substitutions needed to transform one string S_1 into another S_2 . The result is a distance measure d , which can easily be converted into a similarity score s between zero and one by $s = 1 - \frac{d}{d_{max}}$.

We can replace d_{max} , i.e. the maximum distance between two strings, which equals the length of the longer string by $d_{max} = \max(|S_1|, |S_2|)$ regarding the Levenshtein distance.

$$s = 1 - \frac{d}{\max(|S_1|, |S_2|)}$$

As a Bloom filter is a set representation we first convert our strings into sets. This has to be done in a way, that we can later formulate a distance measure upon the constructed set which, loosely spoken, has similar properties as the Levenshtein distance measure.

For this we take q -grams, which are substrings of length q from our string S , that is to be converted into a set. Let n be the number of characters in S and s_i the q -gram starting at position i with $i \in [1, n - q + 1]$. As a result $n - q + 1$ q -grams are generated out of S . If we would use this set to represent a string and measure similarity upon, we would lose the information where the substrings resided, which is important to build our similarity measure. To keep this information we build pairs (i, s_i) consisting of the index i and the q -gram s_i we just described. Further as characters at the beginning and at the end of S are less often used in q -grams, we insert a short string of length $q - 1$ of identical symbols, which are not part of the alphabet of S at the beginning and end of S . See figure 1 on how this is constructed.

Without the extension we would only see the first character in the first q -gram, whereas in the middle of the string each character is found within q q -grams. To also accommodate for shifted parts of the string, resulting from several insert or delete operations, we define an offset o in which we can detect shifted substrings. Based on o each pair (i, s_i) is duplicated $2o + 1$ times forming *positional q -grams*:

$$\forall j \in \{-o, -o + 1, \dots, -1, 0, 1, \dots, o - 1, o\} : (i + j, s_i)$$

For a string S this generates $(2o + 1)(n - q + 1)$ positional q -grams. Our similarity measure uses the set cardinality, denoted as $|A|$ for set A , which is proportional to the length of the extended string. The cardinality of intersected sets $|A \cap B|$ will be the basis for our distance measure.

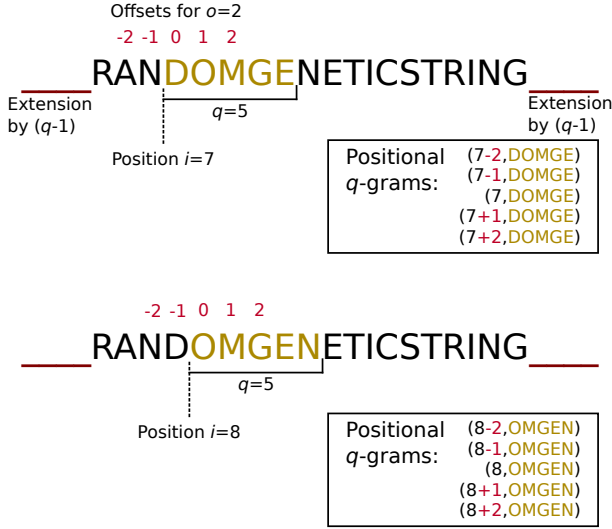


Figure 1: Construction of positional q -grams

As these sets cannot be used directly to compare strings in a privacy-preserving way, which would reveal the original data, we represent them using Bloom filters. A single Bloom filter is used for all positional q -grams generated by a single string S . [14] concludes, that the optimal number of hash functions to do cardinality estimation using Bloom filters is 1. Based on this we fix $k = 1$ and only use a single hash function to build and query Bloom filters throughout the rest of the paper. We also fix the length l of the Bloom filter and the used hash function $h()$ to be equal across all participants.

Determining an appropriate value for l can be done using the formula (2) with the simplification $k = 1$ as introduced above. This results in l being calculated as:

$$l = \frac{-1}{(1-p)^{n-1} - 1} \quad (3)$$

Under these constraints, that k , l and $h()$ are identical, set union \cup and intersection \cap can also be performed upon Bloom filters B_1, B_2 by applying the binary OR or AND operator. Calculating the cardinality of two intersected Bloom filters results in an approximate distance measure due to the probability of mapping two elements to the same filter position through insertion. Our similarity is now defined by $d = |B_1 \cap B_2|$.

Where $d = 0$ means that there are no common elements in both set representations and thus the compared strings are completely distinct regarding our measure. On the other side $d_{max} = (2o+1)(n-q+1)$ is the maximum value that can be achieved and indicates that the shorter string can be found completely within the other for n being the length of the shorter string within the comparison.

3.3 Encrypting the Bloom Filter

We constructed string representations using Bloom filters in section 3.2 and used them to define an appropriate distance measure. However the Bloom filters itself cannot be exchanged between the participants directly, as they can be used to possibly reconstruct the original strings by guessing substrings and checking if they were added to the filter. For

preserving privacy of the filter content, we will make use of an additively homomorphic cryptosystem.

A homomorphic cryptosystem uses at least one homomorphic property to evaluate an operation \oplus on the ciphertext, which translates into applying the equivalent operation $+$ on the plaintext. We will use the additively homomorphic system introduced by Naccache and Stern [15], which is also probabilistic. Let $E(x, r)$ denote the encryption of a value x using a fresh random value r for each encryption, this additively homomorphic system has the following properties:

$$\begin{aligned} E(x, r)E(y, s) &= E(x + y, r + s) \\ E(x, r)^y &= E(xy, ry) \end{aligned}$$

To build an encryption of our Bloom filter B with length l , we encrypt every bit of it separately, storing the resulting l values in a new array C with equal length.

$$\forall i \in [1, l], \text{ fresh } r : C_i = E(B_i, r)$$

This encryption is not to be confused with “encrypted Bloom filter”, which were presented in [16]. Encryption is only performed by Alice, who wants to compare a string privately to one that Bob holds. Bob also slices his string down into positional q -grams, but without the offset multiplication. So instead of having $(2o+1)(n_B - q + 1)$ positional q -grams for his String S_B of length n_B , he just generates $n_B - q + 1$ positional q -grams. These are then added to a new Bloom filter using the previously agreed upon parameters k , l and $h()$.

Alice sends the encryption of her Bloom filter to Bob. Recall that the Bloom filters just contained zeros and ones. So calculating the cardinality of a filter, denoted by $|B|$, can not just be done by counting all bits set to one, but also by calculating the sum over all values $|B| = \sum_{i=1}^l B_i$.

Bob can use this property to calculate the encrypted sum over all values in the encrypted Bloom filter and thus the encrypted cardinality by $Enc(|C|, r) = \sum_{i=1}^l C_i$.

However as Bob is not interested in the encrypted cardinality of Alice’s filter $|B_A|$, he only adds up values at those positions, that are set to one on his own Bloom filter B_B . This is equivalent to building the intersection using binary AND and calculating the resulting cardinality.

$$\forall i, \text{ for which } B_B[i] = 1 : Enc(|B_A \cap B_B|) = \sum C_i$$

This way the encrypted distance $E(d, r)$ for the measure presented in section 3.2 is calculated.

3.4 Privacy-Preserving Similarity

We have calculated an approximate distance value between two strings in section 3.3, that could be returned to Alice for decryption, so she learns the actual computed value. This would however result in another protocol that is prone to the Mastermind attack described in [1]. To mitigate this attack, we restrict the information Alice gains from executing this protocol. Instead of learning the exact result of the comparison, the result is manipulated to give Alice only the information whether the distance is within a previously agreed range, defined by two thresholds t_{min}, t_{max} or not.

Recall that Alice added $(2o+1)(n_A - q + 1)$ positional q -grams to her Bloom filter, while Bob only added $n_B - q + 1$

q -grams. As Bob is calculating the intersection cardinality based on his Bloom filter status, the maximum cardinality or distance measure d that can be achieved equals the number of ones on Bob's filter, its cardinality $|B_B|$. For this the same bits in Alice filter B_A must also be one. The maximum value for $|B_A \cap B_B|$ in our protocol is therefore $d_{max} = n_B - q + 1$.

Now that we know the range of the intersection cardinality as $d \in [0, n_B - q + 1]$, the thresholds must be chosen accordingly with $0 \leq t_{min} \leq t_{max} \leq n_B - q + 1$. Let t_i be a threshold from within the range we just defined $t_i \in [t_{min}, t_{max}]$, with $i \in [0, t_{max} - t_{min}]$ and $E(t_i, r)^{-1}$ be the multiplicative inverse element of $E(t_i, r)$ found through executing the extended euclidean algorithm, which is by the homomorphism definition the encryption of the additively inverse plaintext: $E(t_i, r)^{-1} = E(-t_i, -r)$

By applying the extended euclidean algorithm and a homomorphic addition afterwards, the homomorphic subtraction can be defined.

The protocol for calculating the return values for Alice by Bob is as follows:

- Calculate encrypted inverse thresholds $\forall t_i \in [t_{min}, t_{max}]$, $0 \leq i \leq t_{max} - t_{min} : E(-t_i, r) = E(t_i, r)^{-1}$
- Calculate encrypted threshold differences for all inverse thresholds $D_i = E(d - t_i, r) = E(d, s)E(-t_i, r)$
- Multiply all differences with random values. D_i^r for fresh r drawn uniformly from the plaintext space.

After the first two steps Bob has $t_{max} - t_{min} + 1$ values, expressing the differences between the incremented thresholds and the actual distance. If the calculated distance d is within the defined threshold range $[t_{min}, t_{max}]$, then there is one single element, which is the encryption of zero, due to equal threshold and distance values.

Performing the last step randomizes all values through multiplication with a random number, except the one encrypting a zero. All these $t_{max} - t_{min} + 1$ encrypted values are then shuffled randomly and sent to Alice, who decrypts and checks them against zero. In case a zero is found, she learns that the Bloom filter intersection cardinality was within the specified threshold and thus the compared strings have a distance within that range for our specified metric in section 3.2.

4. SECURITY ANALYSIS

Our protocol is secure under the semi-honest, also called honest-but-curious model and under the assumption the integrated crypto system builds upon. In our case this is based on the higher residuosity problem used in the Naccache-Stern cryptosystem. Other additive homomorphic cryptosystems like Paillier can easily be traded in for the currently used system, bringing possibly another assumption like one based on the decisional composite residuosity problem.

The encryption of the used cryptosystem must however be probabilistic, such that similar plaintexts are mapped to different ciphertexts at random. This is true for our employed Naccache-Stern system and for the Paillier cryptosystem. For system, which do not feature this property, the encrypted Bloom filter sent to Bob only consists of two different values, the encryption of one and zero. Once Bob can associate either one or zero with one of the encrypted values with high probability, he can reconstruct the complete encrypted Bloom filter from Alice.

Following the protocol, both participating parties learn the approximate length of the input sequences, which is necessary to be able to choose an appropriate bloom filter length l and threshold range $[t_{min}, t_{max}]$. In the first part, Alice builds an encrypted bloom filter, only depending on her own sequence and the previously agreed upon bloom filter length, which does not include and thus reveal any information about Bobs sequence. The second step involves Bob calculating on homomorphically encrypted numbers, which reveals no information as the security assumption for the use cryptosystem implies. Bob does not learn the outcome of the protocol, nor anything else than the encrypted bloom filter, thus learning nothing about Alice sequence. As final step, Alice decrypts the randomized values, which are randomly permuted. The information gain about Bobs sequence is, whether the distance d to Bobs sequence lies within the agreed threshold range $[t_{min}, t_{max}]$, so she learns a single bit of information.

5. EVALUATION

For our experiments we used a Linux Laptop with an Intel Core2 Duo T9600 running at 2.8 GHz. The code is written in Java, using the Bouncy Castle library¹. First we run tests to evaluate the relation between the Levenshtein distance and the measure introduced in section 3.2. Further we evaluate the runtime performance of our algorithm for string lengths, which were also used for comparing other privacy-preserving string matching protocols.

5.1 Distance Measure

As we compared our similarity metric to the Levenshtein distance while constructing it, we now measure the influence of the edit distance on our metric. The sliding window offset is set to $o = 11$, where the length of used substrings is $q = 23$. To run the test we set a Bloom filter up to use a single hash function, in our case "SHA1" modulo the size of the filter. We use strings containing roughly $n = 1000$ characters, which means we have about $m = (2o + 1)(n - q + 1) = 22494$ positional q -grams to be inserted for each Bloom filter. The probability that a false-positive test occurs after the m elements are added to the filter is set to $p = 0.1$, which in turn generates a Bloom filter of size 217862 bits. We used 1000 runs and for each applied a fixed number of edit operations. The two resulting strings per run are then compared using our distance measure. The resulting value is the intersection cardinality of both Bloom filters B_1 and B_2 . This represents the number of identical elements and thus correlates to the number of identical characters in both strings. To get to the difference between the strings, we subtract this similarity value from the maximum number of set elements $(2o + 1)(n - q + 1)$.

This difference should correlate to the Levenshtein distance and approximate it. Figure 2 shows a Boxplot for every Levenshtein distance and the according 1000 runs tested with our approximate distance measure. As can be seen from the figure, our distance value approximates small Levenshtein distances very good, with a narrow range of possible values and a small variance. In our figure up to an edit distance of 9 the values between

quartiles Q_1 and Q_3 are essentially different between two neighbouring Levenshtein distances. Further more the me-

¹<http://www.bouncycastle.org/>

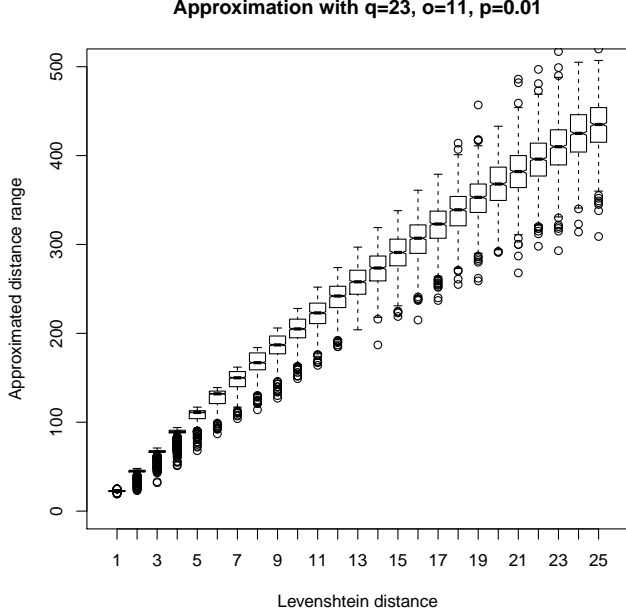


Figure 2: Approximation of small Levenshtein distances by our distance measure

dian distances across different Levenshtein distances are significantly different, as represented by the notches, even up to the highest tested Levenshtein distance 100. The Pearson correlation between the Levenshtein distances and the mean approximated distances is $c_p = 0.962$ for up to 100 edit operations and $c_p = 0.997$ for up to 25 edits.

We can also see that the variance in our approximated distances grows with more edits and the means are not growing linear, but on a diminishing scale. In return approximations for larger Levenshtein distances are less accurate than for smaller ones.

The outlying circles at the upper end of a Boxplot represent mainly those strings, where the difference in length between the compared sequences is above o , which means that string matching using our sliding window of size o works up to the point where the absolute value of insert versus delete operations is smaller or equals o . Once the difference in length grows beyond that point, similar sequences may not be detected anymore, as the offset is too small so two identical q -grams are not mapped to the same element in the Bloom filter anymore.

5.2 Protocol Execution Time

To evaluate the performance of our protocol, we ran 1000 runs for each test. The parameters were set to $q = 5$, $s = 2$, $p = 0.1$ and edit distances for up to 5 were used.

The client runtime depends linearly on the length of the input sequence, where the most time is spent on decrypting the results from the server and encrypting the Bloom filter prior to transmission. We can see a pretty high variance on client runtimes, growing linearly with longer sequences, due to the unknown number of results which are needed to be decrypted until a zero is found. If the distance between both compared sequences is not within the predefined range

of thresholds, the client always needs to decrypt all results, always taking the maximum time to decrypt, as no zero will be found within the returned values.

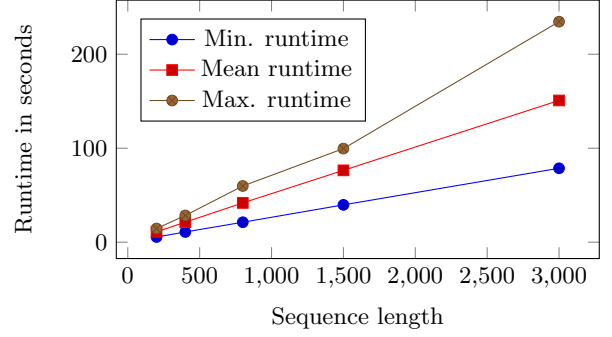


Figure 3: Client runtimes

Server runtime only depends on the sequence length, so runtimes for equal lengths are virtually identical. Variance across runtimes of equal length is near zero.

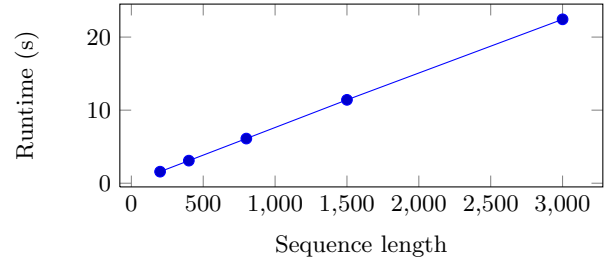


Figure 4: Server runtimes

The amount of data that needs to be transferred between Alice and Bob is shown in table 1 and grows linearly with the length of the Bloom filter for the traffic from Alice to Bob and linearly with the size of the threshold range for the traffic from Bob to Alice. For this test the maximum threshold t_{max} was set to the maximum similarity $d_{max} = (n - q + 1)$, the minimum threshold t_{min} was set to $0.75 * d_{max}$. This way the bandwidth needed for transmitting the results also grows linearly with the Bloom filter size, which in return depends on the length of the compared strings.

Sequence length	Client to server	Server to client
200	472KB	8.18KB
400	933KB	15.85KB
800	1855KB	31.18KB
1500	3464KB	58.01KB
3000	6918KB	115.51KB
10000	23011KB	383.81KB

Table 1: Bandwidth used for transmission

Communication complexity from client to server is $O(l)$, with l being the bloom filter size, which in return is calculated as described in equation (3). l grows linear with the sequence length n . The answer from the server has a complexity of $O(r)$, with r being the range size of the thresholds $r = t_{max} - t_{min}$, as there are r encrypted values being transmitted back to the client. In our example

r grows linearly with the sequence length n . The overall communication complexity thus amounts to $O(l + r)$.

As for the computational complexity, the client first needs to transform its sequence into positional q-grams through the extended string, as described in section 3.2. This results in $n + q - 1$ positional q-grams, with q being a small constant. To include insert and delete operations we added an offset of o in every direction for every character, thus generating $(n + q - 1)(2o + 1)$ positional q-grams which need to be added to the bloom filter of length l , which in return needs to be encrypted. The client has for its first step a computational complexity of $O(n * o + l)$. The encryption of the bloom filter elements is by far the most demanding operation for the client.

Looking at the complexity of the server's computation, we have the generation of a bloom filter without the offset extension, thus adding $n + q - 1$ elements to a bloom filter. Then, we homomorphically add about $p * l$ values to calculate the bloom filter intersection, with $p = 0.1$, subtract r values from the result and multiply every difference with a random value. As a result we get an overall computational complexity for the server of $O(n + q + p * l + 2r)$, with $p < 1$. As a last step the client decrypts the r results with $O(r)$.

Comparing these results to the ones given by Jha [11] and Huang [17] in the evaluations of their state of the art protocols, we achieve an increased performance starting with the smallest string lengths of 200 characters. While the referenced protocols have computational complexity of $O(n \log n)$, $O(n^2)$ and $O(n * m)$ for input string lengths n and m , our protocol has linear complexity. Therefore, matching of larger strings is also much more efficient.

6. CONCLUSION

We presented a novel, non-interactive approach for a privacy-preserving approximate string matching protocol, that is build to mitigate a recent, generic attack based upon the Mastermind game, described in [1]. An attacker will no longer learn exact values or approximations, but only if two compared strings are within a predefined closeness range.

Due to the computation having linear complexity in the used sequence length and the communication having linear complexity in the range of allowed distances, respectively in the Bloom filter length, this protocol is very practical and was tested for strings of up to 10000 characters in length, for which about 570 seconds were required to run a comparison on the mentioned hardware.

Further enhancements upon this protocol can be done by turning it into an approximate protocol that reveals more information to the client than the current approach does and evaluate upon that how this influences the vulnerability for the mentioned attack. Additional discussion and evaluation could go into using our protocol for database searches.

References

- [1] M. T. Goodrich, "The Mastermind Attack on Genomic Data," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, May 2009, pp. 204–218.
- [2] R. W. Hamming, "Error-Detecting and Error-Correcting Codes," *Bell System Technical Journal*, vol. 29, pp. 147–160, 1950.
- [3] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences." *Journal of molecular biology*, vol. 147, no. 1, pp. 195–7, Mar. 1981.
- [4] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing." *Briefings in bioinformatics*, vol. 11, no. 5, pp. 473–83, Sep. 2010.
- [5] P. A. V. Hall and G. R. Dowling, "Approximate String Matching," *ACM Computing Surveys*, vol. 12, no. 4, pp. 381–402, Dec. 1980.
- [6] T. Bachteler and R. Schnell, "An empirical comparison of approaches to approximate string matching in private record linkage," *Proceedings of Statistics Canada*, 2010.
- [7] R. Schnell, T. Bachteler, and J. Reiher, "Privacy-preserving record linkage using Bloom filters." *BMC medical informatics and decision making*, vol. 9, no. 1, p. 41, Jan. 2009.
- [8] Y. Huang, D. Evans, and J. Katz, "Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?" *NDSS*, 2012.
- [9] E. D. Cristofaro, P. Gasti, and G. Tsudik, "Fast and Private Computation of Cardinality of Set Intersection and Union," *Cryptology ePrint Archive, Report 2011/141*, pp. 1–19, 2011.
- [10] J. R. Troncoso-Pastoriza, S. Katzenbeisser, and M. Celik, "Privacy preserving error resilient dna searching through oblivious automata," in *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*. New York, New York, USA: ACM Press, Oct. 2007, p. 519.
- [11] S. Jha, L. Kruger, and V. Shmatikov, "Towards Practical Privacy for Genomic Computation," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, May 2008, pp. 216–230.
- [12] P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik, "Countering GATTACA: efficient and secure testing of fully-sequenced human genomes," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 691–702.
- [13] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.
- [14] O. Papapetrou, W. Siberski, and W. Nejdl, "Cardinality estimation and dynamic length adaptation for Bloom filters," *Distributed and Parallel Databases*, vol. 28, no. 2-3, pp. 119–156, Sep. 2010.
- [15] D. Naccache and J. Stern, "A new cryptosystem based on higher residues," *Proceedings of the 5th ACM conference on computer and communication security*, pp. 59–66, 1998.
- [16] S. M. Bellovin, S. M. Bellovin, and W. R. Cheswick, "Privacy-Enhanced Searches Using Encrypted Bloom Filters," 2004.
- [17] Y. Huang, D. Evans, and J. Katz, "Faster secure two-party computation using garbled circuits," *USENIX Security Symposium*, 2011.